The Basics of UNIX/Linux

12-2. Pointer parameters and Dynamic arrays

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

datalab

data science laboratory

This file is based on slides from Justin Hsia.

Lecture Outline

- Pointers as Parameters

- dynamically allocated 2D arrays
  - Method1:
    - Allocate a single chunk of NxM heap space
  - Method2:
    - Allocate an array of arrays: allocate 1 array of N pointers to arrays, and allocate N M bucket array of values (on for each row).

Passing pointers to functions

- Passing pointers to functions
  - Allows the referenced object to be accessible in multiple functions without making the object global
  - If the data needs to be modified in a function
    - it needs to be passed by a pointer
  - When the data is a pointer that needs to be modified,
    - then we pass it as a pointer to a pointer

C is Call-By-Value

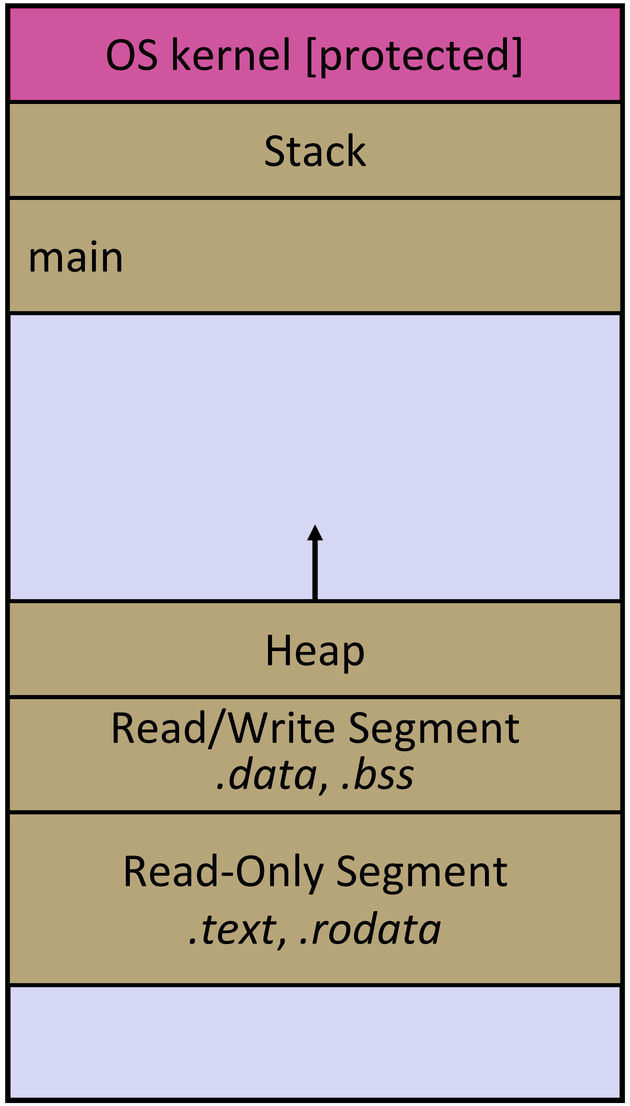
- C (and Java) pass arguments by *value*
  - Callee receives a **local copy** of the argument
    - Register or Stack
  - If the callee modifies a parameter, the caller's copy *isn't* modified

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```

Broken Swap

brokenswap.c

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



Faking Call-By-Reference in C

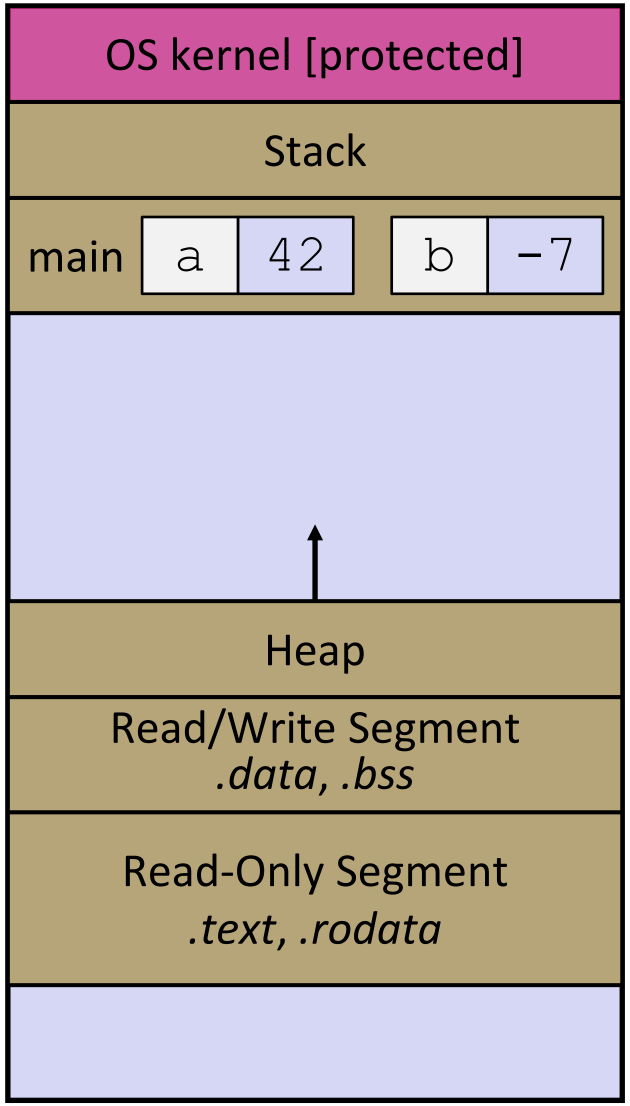
- Can use pointers to *approximate* call-by-reference
  - Callee still receives a **copy** of the pointer (*i.e.* call-by-value), but it can modify something in the caller's scope by dereferencing the pointer parameter

```c
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(&a, &b);
  ...
```

Fixed Swap

swap.c

```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(&a, &b);
  ...
```

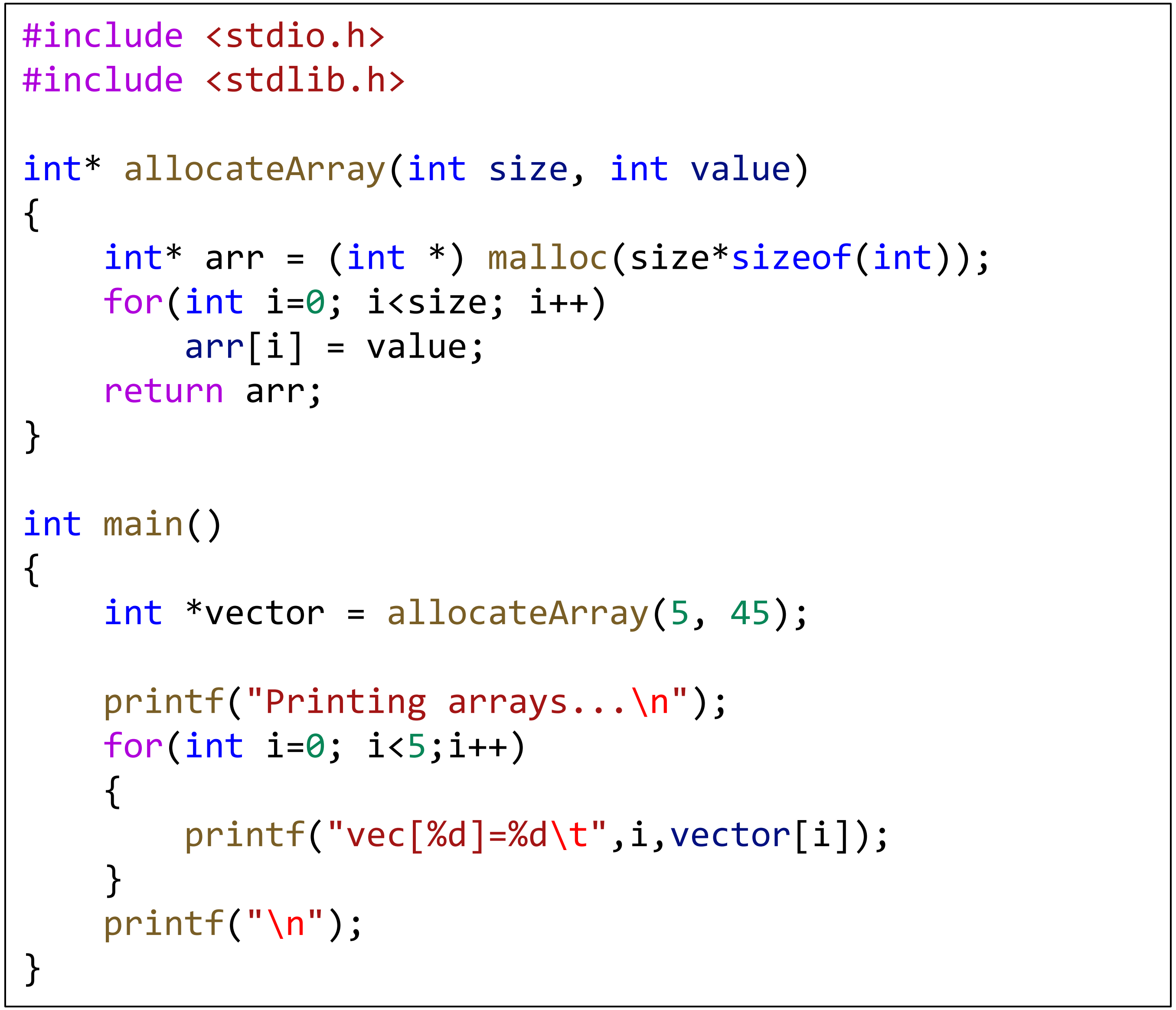


Returning a pointer

- Simply declare the return type to be a pointer to the appropriate data type

- If we need to return an object from a function
  - (1) allocate memory within the function using malloc and return its address.
    - Caller is responsible for deallocating the memory returned.
  - (2) Pass an object to function where it is modified.
    - Caller is responsible for allocation and deallocation of the object’s memory

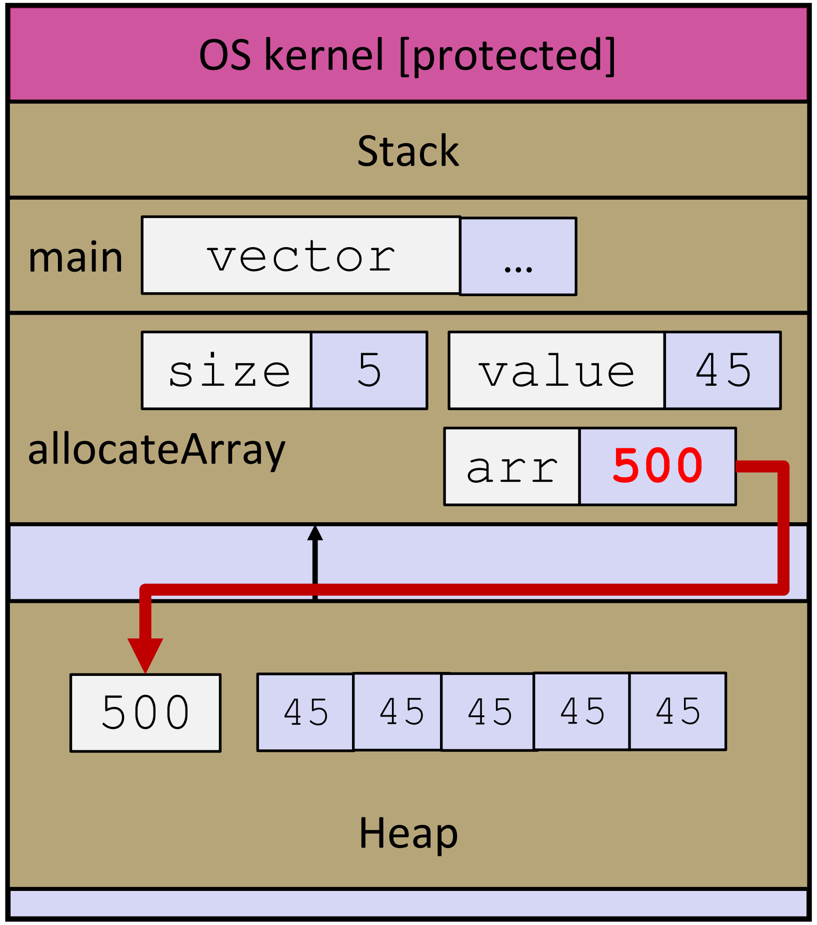
Ex1: allocArray.c (1/2)



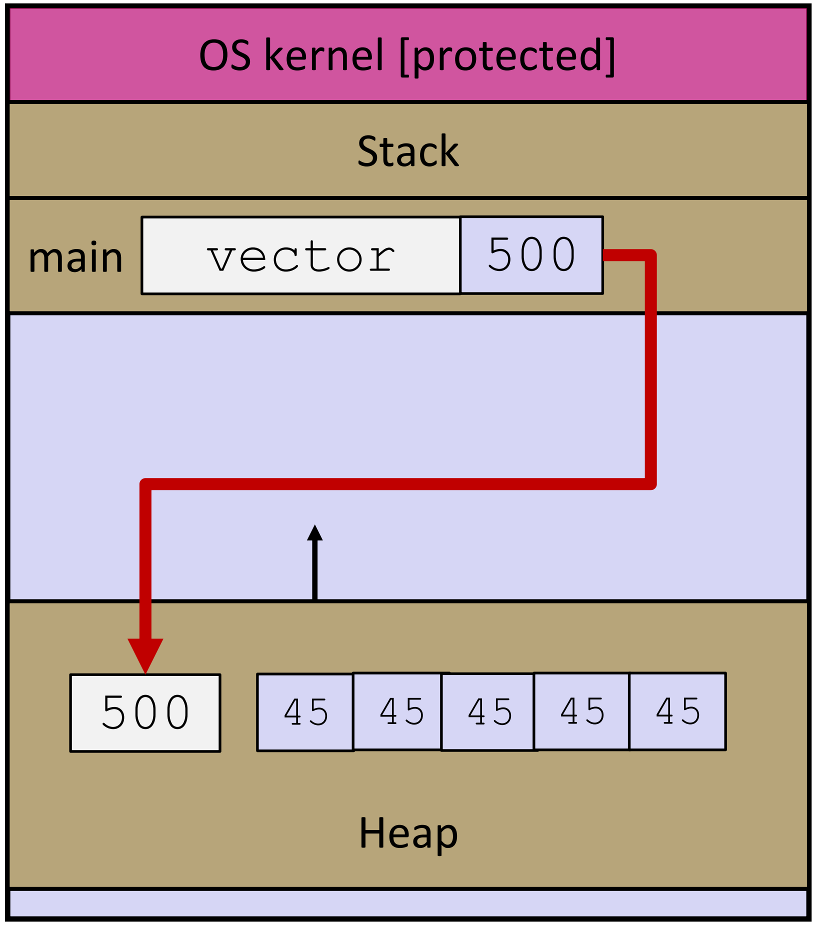
```c
#include <stdio.h>
#include <stdlib.h>

int* allocateArray(int size, int value)
{
    int* arr = (int *) malloc(size*sizeof(int));
    for(int i=0; i<size; i++)
        arr[i] = value;
    return arr;
}

int main()
{
    int *vector = allocateArray(5, 45);

    printf("Printing arrays...\n");
    for(int i=0; i<5;i++)
    {
        printf("vec[%d]=%d\t",i,vector[i]);
    }
    printf("\n");
}
```

Ex1: allocArray.c (2/2)

- Right before the return statement is executed



- After the function has returned



Potential problems

- Returning an uninitialized pointer
- Returning a pointer to an invalid address
- Returning a pointer to a local variable
- Returning a pointer but failing to free it
  - The caller is responsible for deallocating it

```
int *vector = allocateArray(5, 45);
 …
 free(vector);
```

Allocate Array version2 (1/2)

- allocArrayv2.c
  - Passing a simple pointer
  - Not returning a pointer

```
#include <stdio.h>
#include <stdlib.h>

void allocateArray(int *arr, int size, int value)
{
    arr = (int *) malloc(size*sizeof(int));
    if( arr!= NULL)
    {
        for(int i=0; i<size; i++)
            arr[i] = value;
    }
}

int main()
{
    int *vector = NULL;
    allocateArray(&vector, 5, 45);

    printf("Printing arrays...\n");
    for(int i=0; i<5;i++)
    {
        printf("vec[%d]=%d\t",i,vector[i]);
    }
    printf("\n");
}
```

Allocate Array version2 (2/2)

- Compile

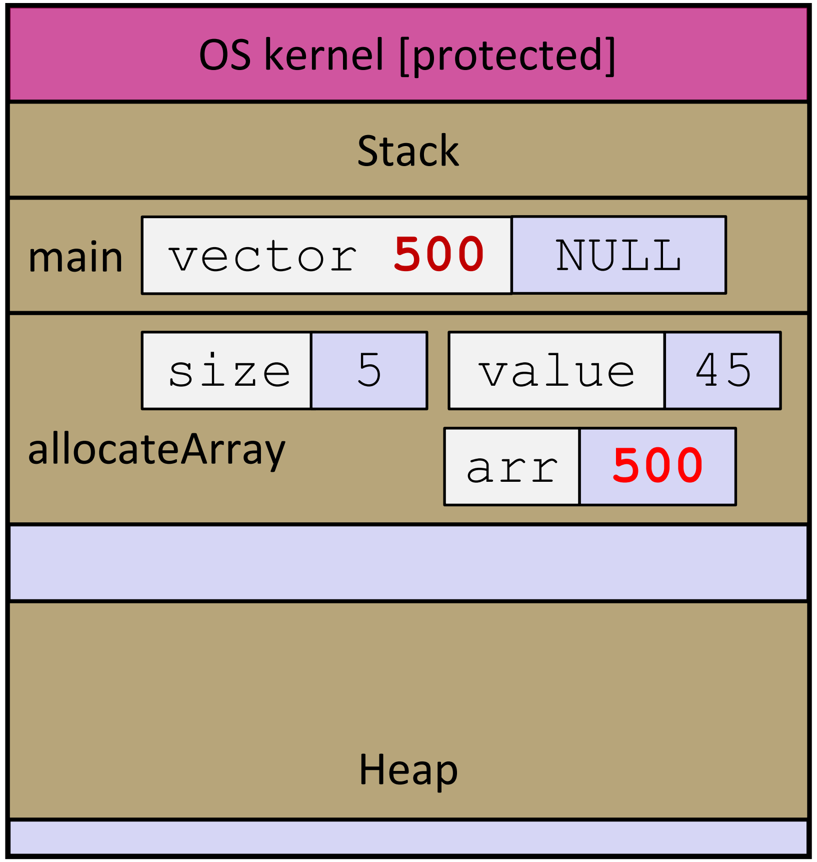
```
$ gcc –o aav2 allocArrayv2.c
alloc_ex2.c: In function 'main':
alloc_ex2.c:17:19: warning: passing argument 1 of 'allocateArray’
              from incompatible pointer type [-Wincompatible-pointer-types]
    allocateArray(&vector, 5, 45);
                  ^
alloc_ex2.c:4:6: note: expected 'int *' but argument is of type 'int **'
 void allocateArray(int *arr, int size, int value)
      ^~~~~~~~~~~~~
```

- Execute

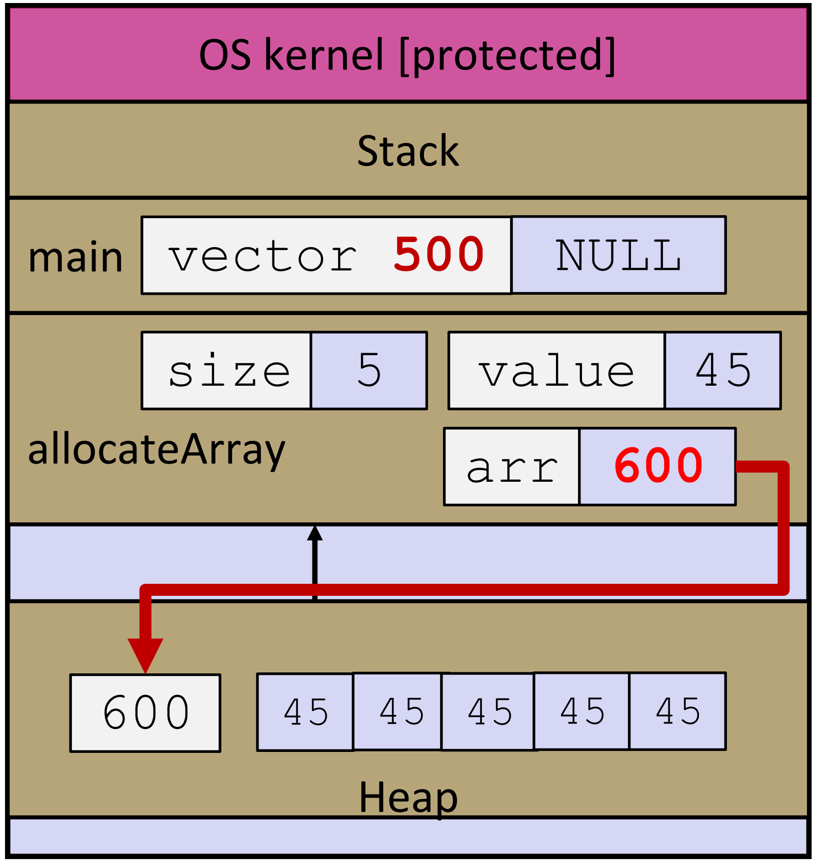
```
$ ./aav2
Printing arrays...
Segmentation fault (core dumped)
```

why version2 is not working? (1/2)

- Before malloc
  - arr contains 500, which was passed to it from vector

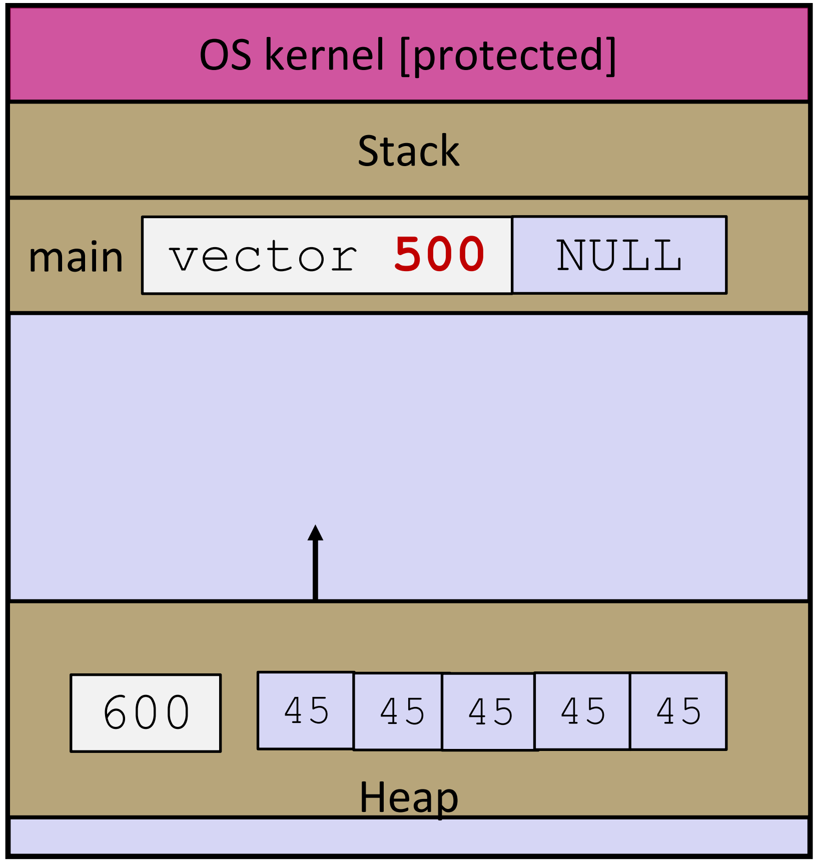


- After malloc
  - arr has been modified to point to a new place in the heap



why version2 is not working? (2/2)

- After the function returns



OS kernel [protected]

Stack

main vector 500 NULL

600 45 45 45 45 45

Heap

```
#include <stdio.h>
#include <stdlib.h>

void allocateArray(int *arr, int size, int value)
{
    arr = (int *) malloc(size*sizeof(int));
    if( arr!= NULL)
    {
        for(int i=0; i<size; i++)
            arr[i] = value;
    }
}

int main()
{
    int *vector = NULL;
    allocateArray(&vector, 5, 45);

    printf("Printing arrays...\n");
    for(int i=0; i<5;i++)
    {
        printf("vec[%d]=%d\t",i,vector[i]);
    }
    printf("\n");
}
```

Passing a pointer to a pointer

- When a pointer is passed to a function,
  - it is passed by value.
- If we want to modify the original pointer and not the copy of the pointer
  - We need to pass it as a pointer to a pointer

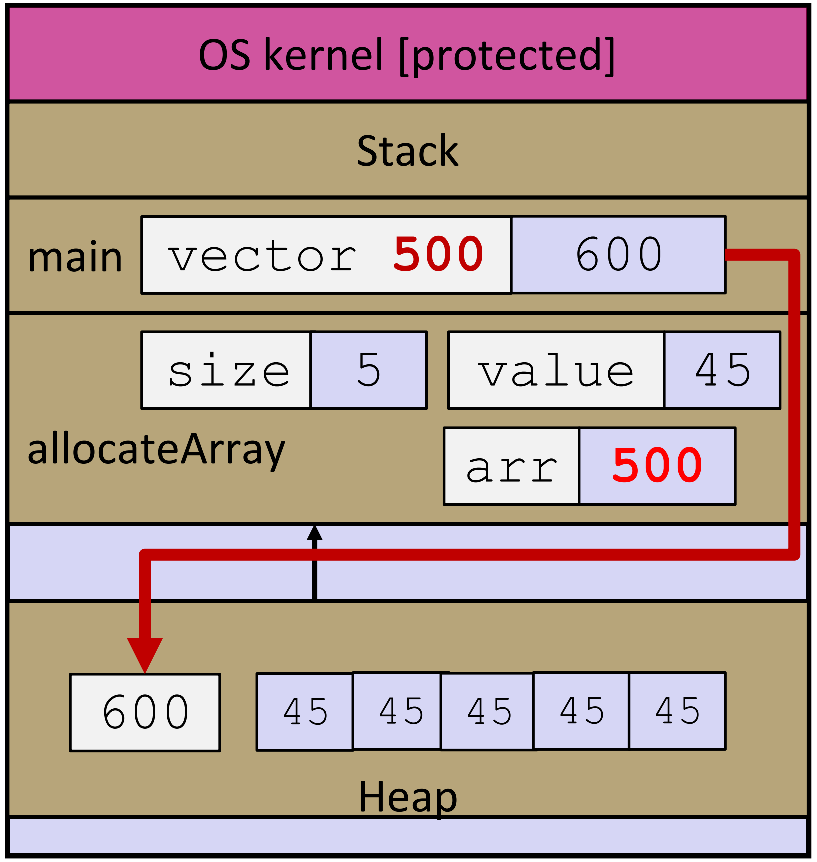
Allocate Array version3: good (1/2)

- Pparray.c
  - Pass a **pointer to integer array**
  - Return the allocated memory back through the first parameter

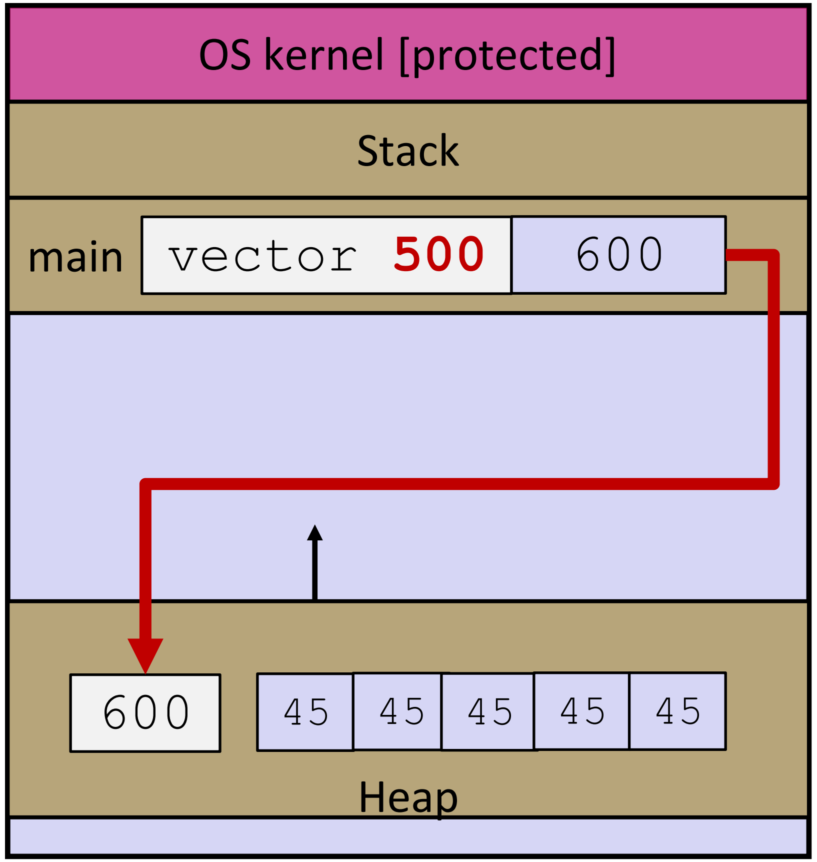
```
#include <stdio.h>
#include <stdlib.h>

void allocateArray(int **arr, int size, int value)
{
    *arr = (int *) malloc(size*sizeof(int));
    if( *arr!= NULL)
    {
        for(int i=0; i<size; i++)
            *(*arr+i) = value;
    }
}

int main()
{
    int *vector = NULL;
    allocateArray(&vector, 5, 45);

    printf("Printing arrays...\n");
    for(int i=0; i<5;i++)
    {
        printf("vec[%d]=%d\t",i,vector[i]);
    }
    printf("\n");
}
```

Allocate Array version3: good (2/2)

- After malloc returns and array is initialized



- After function returns



Lecture Outline

- **Pointers as Parameters**
- **dynamically allocated 2D arrays**
  - **Method1:**
    - **Allocate a single chunk of NxM heap space**
  - **Method2:**
    - **Allocate an array of arrays: allocate 1 array of N pointers to arrays, and allocate N M bucket array of values (on for each row).**

Method 1: the memory efficient way

- a single NxM malloc
  - really this is a large 1-dim array of int values onto which we will map 2D accesses

```
void init2D(int *arr, int rows, int cols) {
    int i,j;
    for(i=0; i < rows; i++) {
        for(j=0; j < cols; j++) {
            arr[i*cols +j] = 0;
        }
    }
}

int main() {
    int *array;
    array = malloc(sizeof(int)*N*M);
    if(array != NULL) {
        init2D(arr, N, M);
    }
    // do anything yow want to
}
```

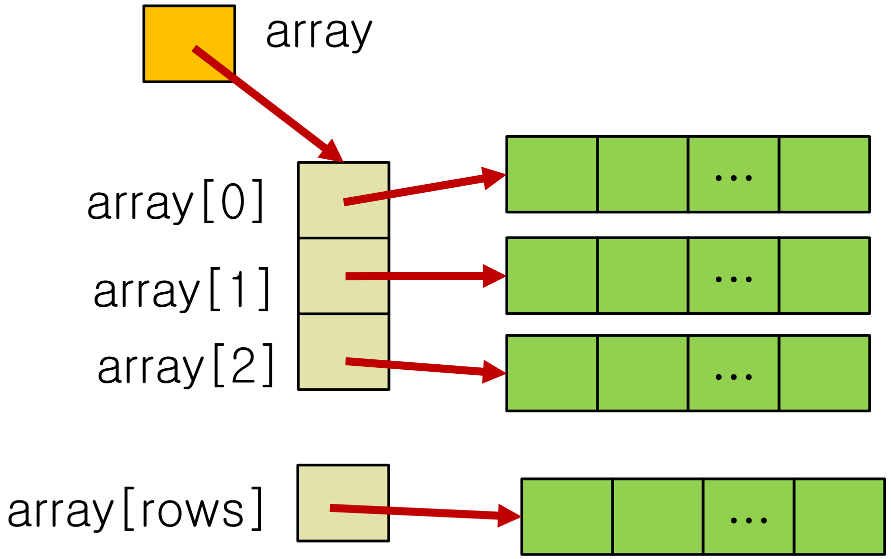
Method2: array of pointers

- the "can still use [r][c] syntax to access" way
  - N mallocs, one for each row, plus one malloc for array of row arrays

```
int main() {
    // an array of int arrays (a pointer to pointers to ints)
    int **array;
    // allocate an array of N pointers to ints
    // malloc returns the address of this array (a pointer to (int *)'s)
    array = (int **)malloc(sizeof(int *)*ROWS);
    // for each row, malloc space for its buckets and add it to
    // the array of arrays
    for(int i=0; i < ROWS; i++) {
        array[i] = (int *)malloc(sizeof(int)*COLS);
    }
    // Use current time as seed for random generator
    srand(time(0));
    init2DArrayRandom(array, ROWS, COLS);
    print2DArray(array,ROWS,COLS);
}
```

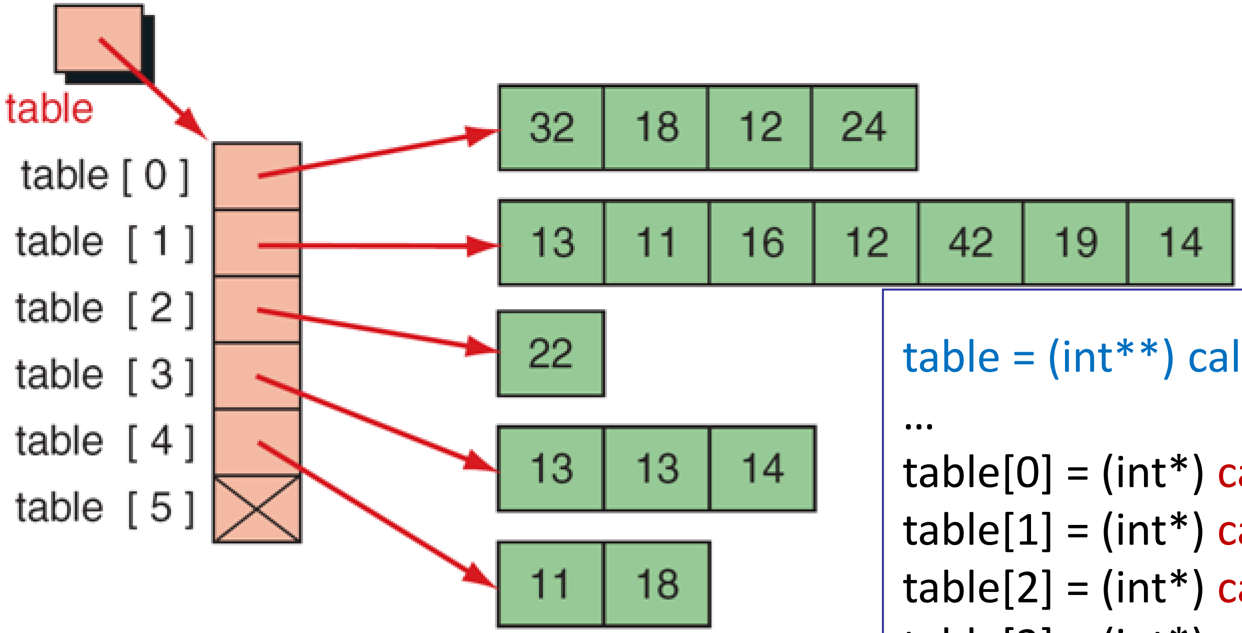
Method2: array of pointers

```
#define ROWS 5
#define COLS 3

void init2DArrayRandom(int **arr, int rows, int cols) {
    int i,j;

    for(i=0; i < rows; i++) {
        for(j=0; j < cols; j++) {
            arr[i][j] = rand()%10000;
        }
    }
}

void print2DArray(int **arr, int rows, int cols)
{
    int i,j;
    for(i=0; i < rows; i++) {
        for(j=0; j < cols; j++) {
            printf("[%d,%d]: %d\t",i,j, arr[i][j]);
        }
        printf("\n");
    }
}
```



A Dynamic 2D Array

- Dynamic 2D Array
  - A dynamic array is an array of pointers to save space when not all rows of the array are full.



```
table = (int**) calloc (rowNum +1, sizeof(int*) );
…
table[0] = (int*) calloc(4, sizeof(int));
table[1] = (int*) calloc(7, sizeof(int));
table[2] = (int*) calloc(1, sizeof(int));
table[3] = (int*) calloc(3, sizeof(int));
table[4] = (int*) calloc(2, sizeof(int));
table[5] = NULL;
```

Q&A